Распространённые ошибки изменения схемы базы данных PostgreSQL

Николай Самохвалов
nik@postgres.ai





Popular PostgreSQL schema migration failures

Nikolay Samokhvalov
nik@postgres.ai



Speaker: Nikolay Samokhvalov

- Database systems:
  - 2002-2005: 

  - since 2005: 

- Worked on XML data type and functions (2005-2007)
- Long-term community activist – #RuPostgres, Postgres.tv
- Conferences Program Committee 





etc.
- Current business: 

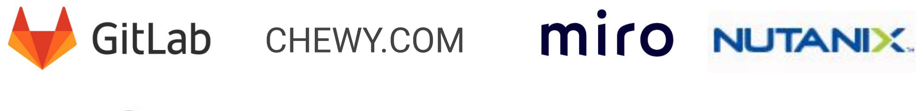


👉 *Created/reviewed more than* ***1,000 DB migrations***

Postgres.ai

– clone DB of any size in a few seconds in bring them in any point of the DevOps lifecycle

- automated (in CI) testing of DB migrations
- guess-free SQL optimization
- Instantly deploy full-size staging apps





Fresh version of these slides

bit.ly/highload2021

– comments are open (and welcome!)



This talk's goals

- 😶 see *some* examples of mistakes, horror stories
- 😶 learn something new

This talk's goals

😶 see *some* examples of mistakes, horror stories

😶 learn something new

✅ how avoid downtime and issues – learn *principles*

✅ see concrete path to having downtime-free process

Terminology

DML – database manipulation language
(SELECT / INSERT / UPDATE / DELETE, etc.)

DDL – data definition language

(CREATE …, ALTER …, DROP …)

DB migrations – planned, incremental changes
of DB schema and/or data

DB schema migration & data migration
DB schema evolution, schema versioning
DB change management, and so on



Applying a schema migration to a production database is always a risk

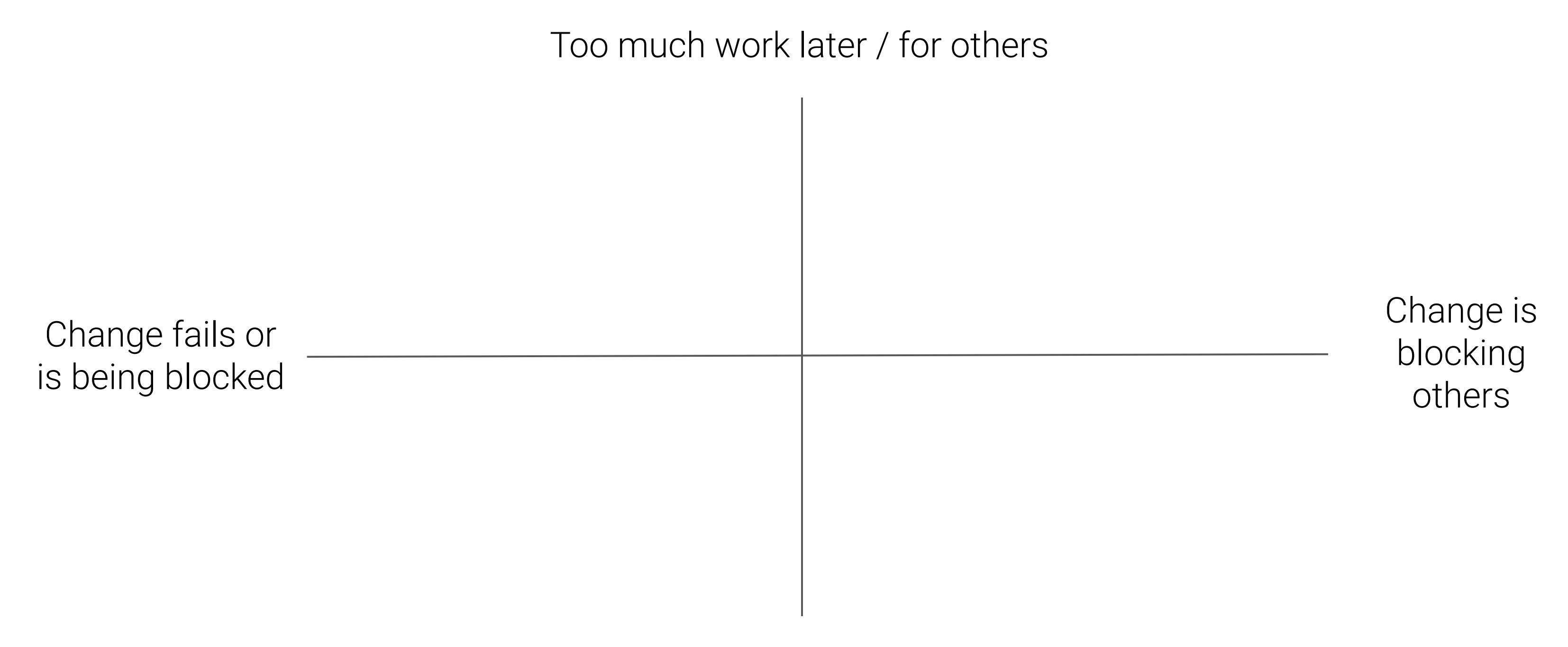
Wikipedia

https://en.wikipedia.org/wiki/Schema_migration

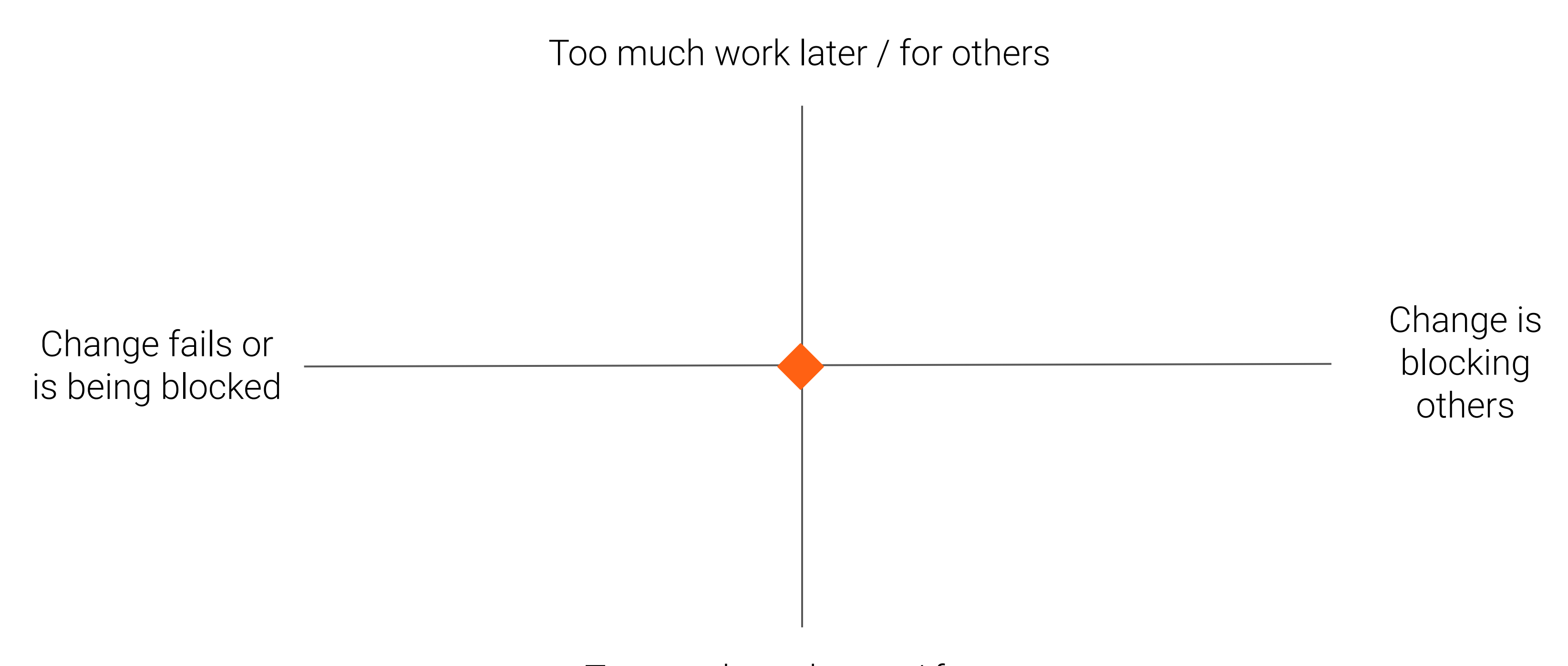
Types of mistakes

1. Schema mismatch
2. Heavy operation (processing too much data)
3. Blocked (cannot acquire lock)
4. Blocker (holding heavy lock)
5. Post-deployment issues

DB change – risk classification

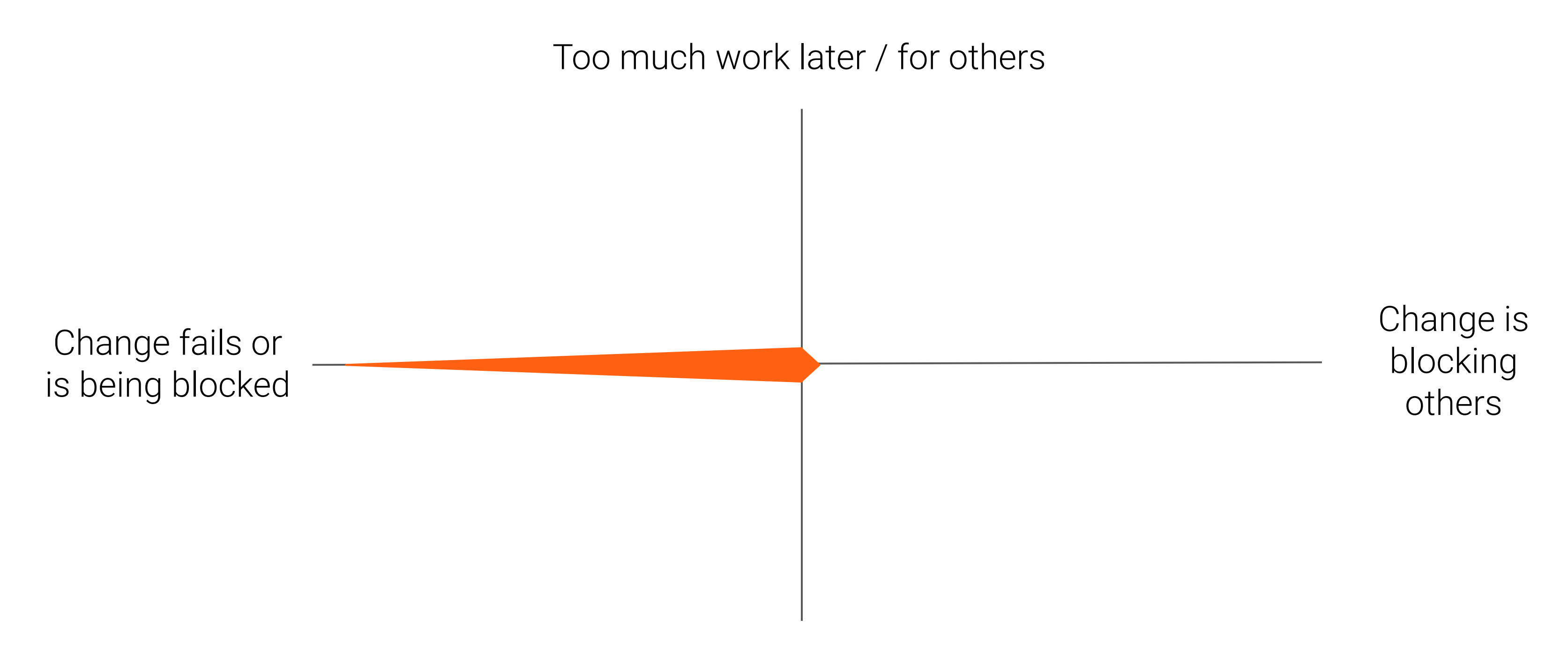


Ideal Change

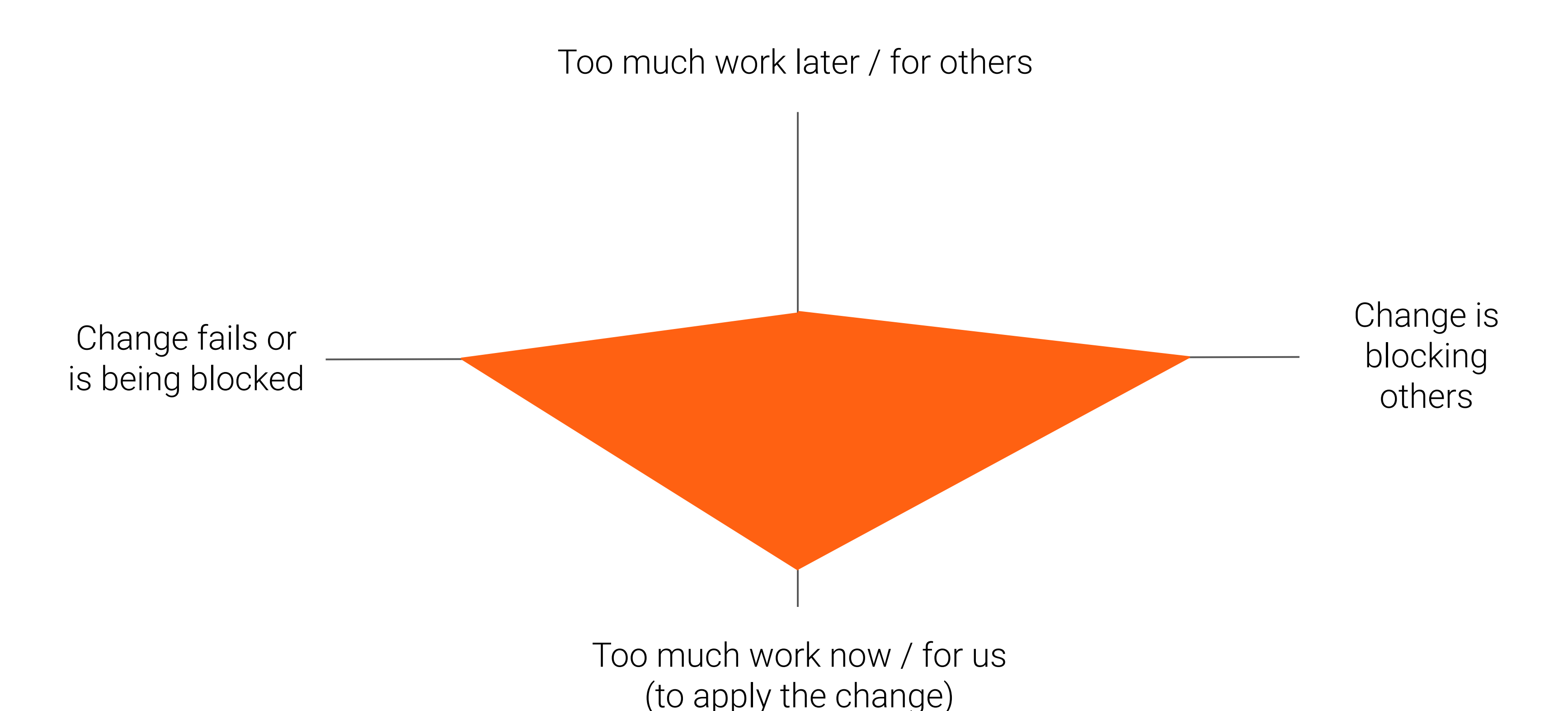


Postgres.ai

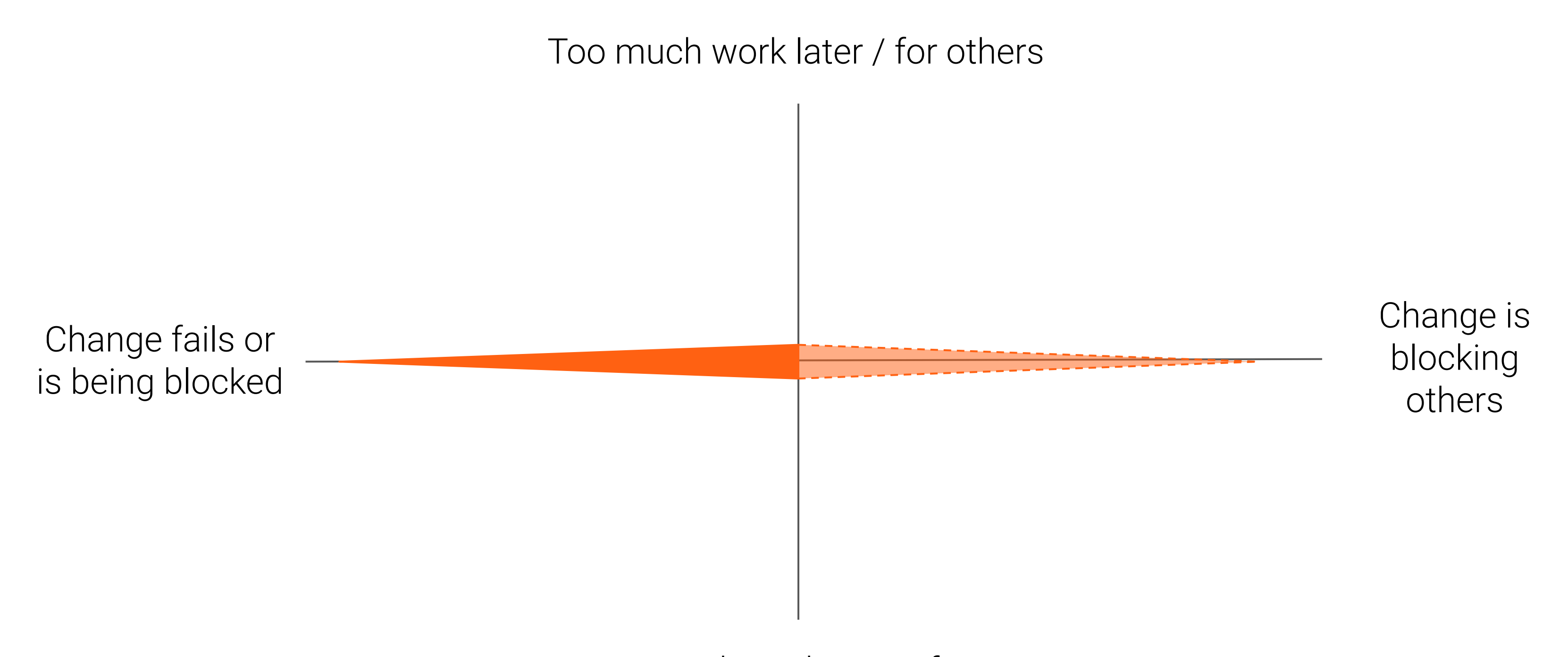
Schema mismatch



Heavy operation

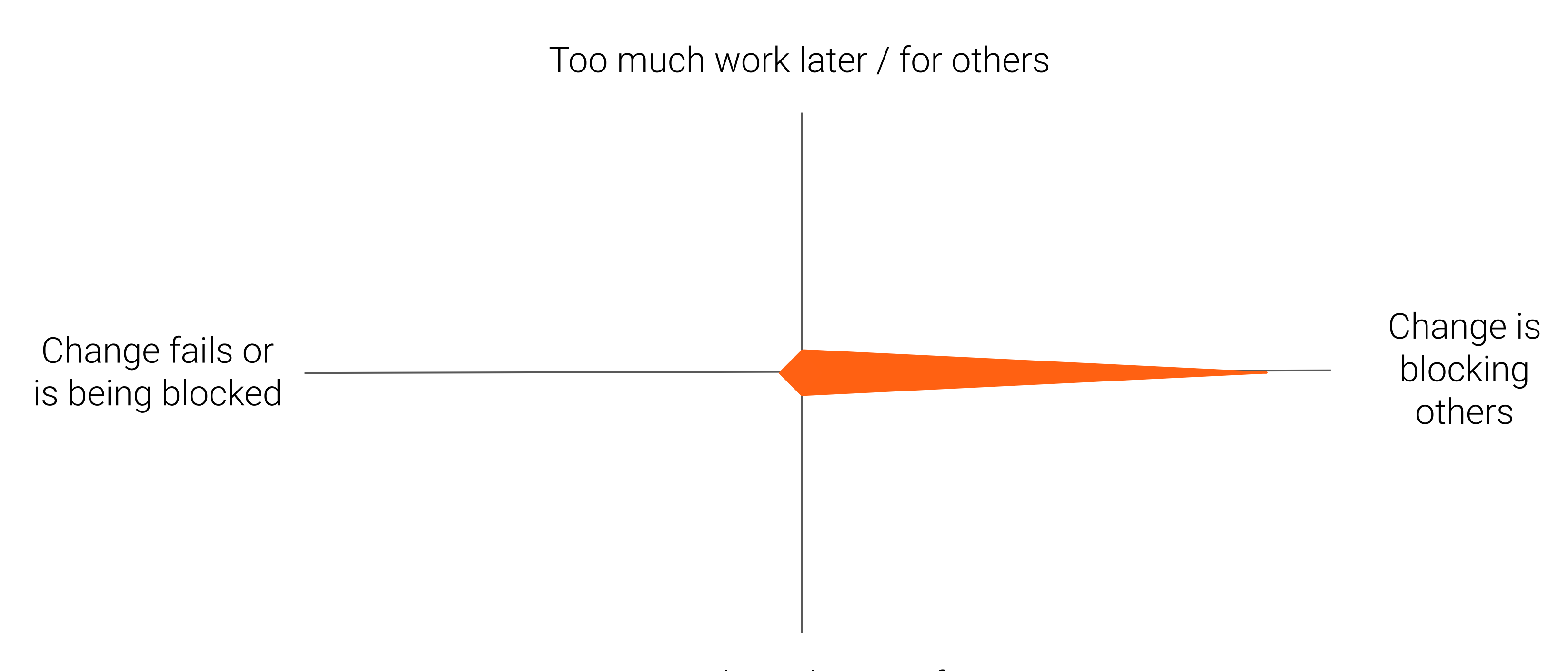


Blocked (cannot acquire lock)

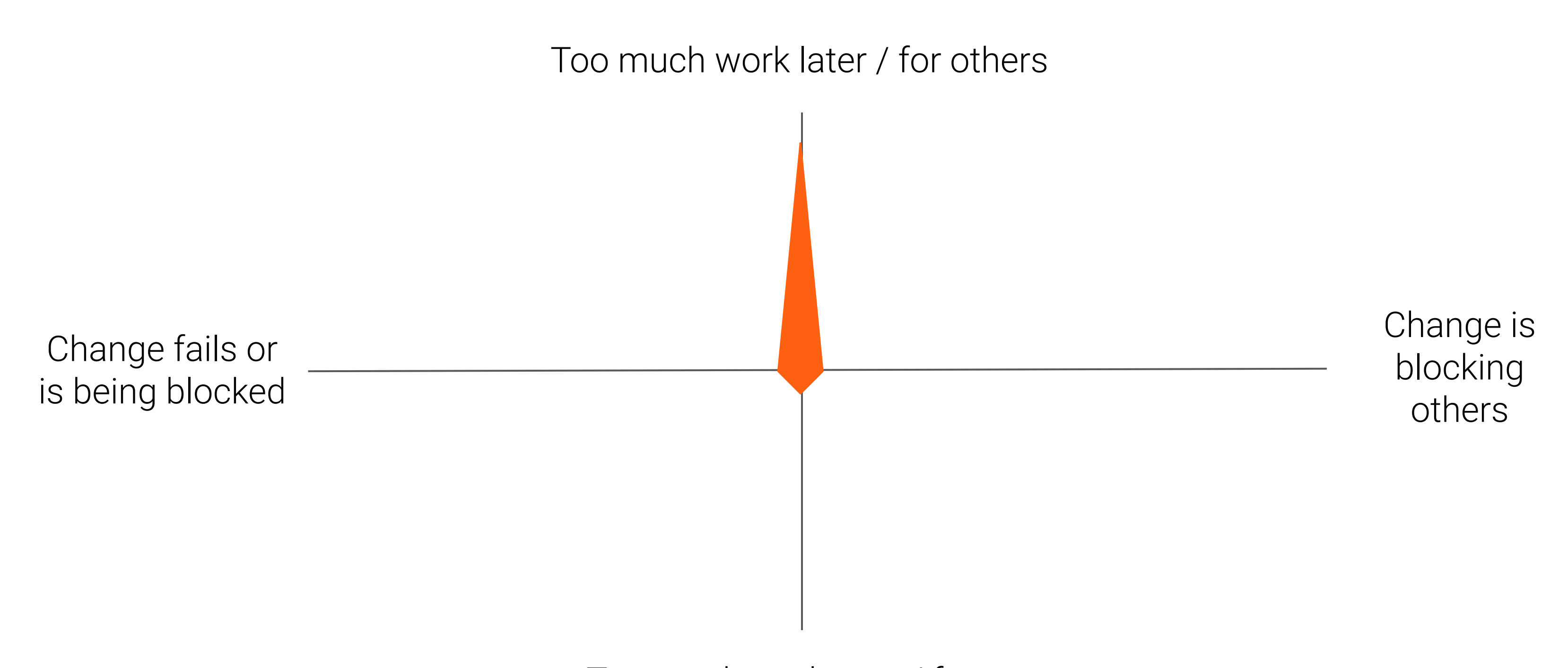


Postgres.ai

Blocker (holding heavy lock)

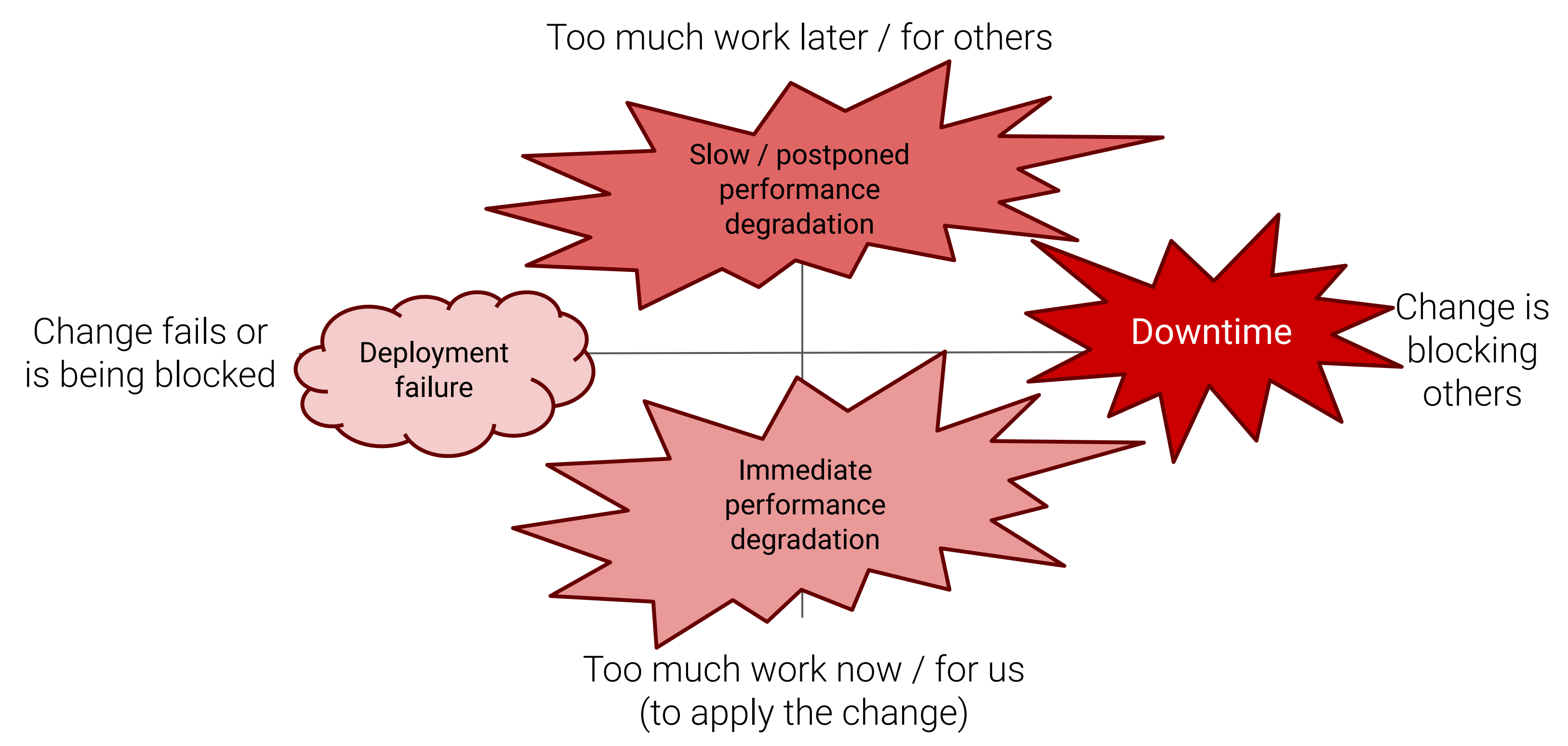


Post-deployment issues



Postgres.ai

DB changes – risk classification



Example #1

```
create table t1 (
  id int primary key,
  val text
);

-- dev, test, QA, staging, whatever – OK

-- prod:
ERROR:  relation "t1" already exists
```

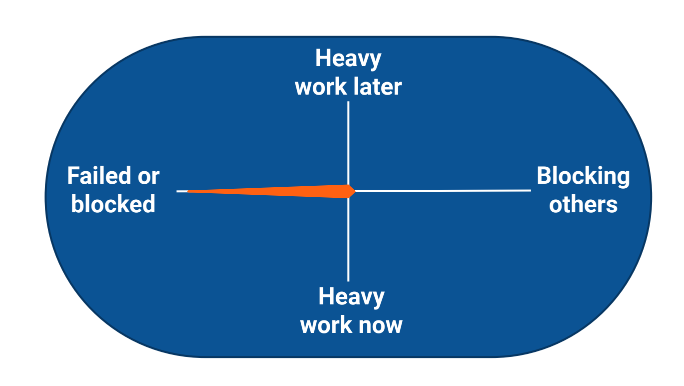


Example #1

```
create table t1 (
  id int primary key,
  val text
);


-- dev, test, QA, staging, whatever – OK


-- prod:
ERROR:  relation "t1" already exists
```





IF [NOT] EXISTS

```
create table if not exists t1 (
  id int primary key,
  val text
);
```

```
NOTICE:  relation "t1" already exists, skipping
```

```
CREATE TABLE
```



Start using DB schema migration tool













Test changes in CI

- Both DO and UNDO steps are supported (can revert)
- CI: test them all
  - Better: DO, UNDO, and DO again

Test changes in CI

- Both DO and UNDO steps are supported (can revert)
- CI: test them all
    - Better: DO, UNDO, and DO again

Now guess what…

“Thanks” to `IF NOT EXISTS`, we now may leave UNDO empty!



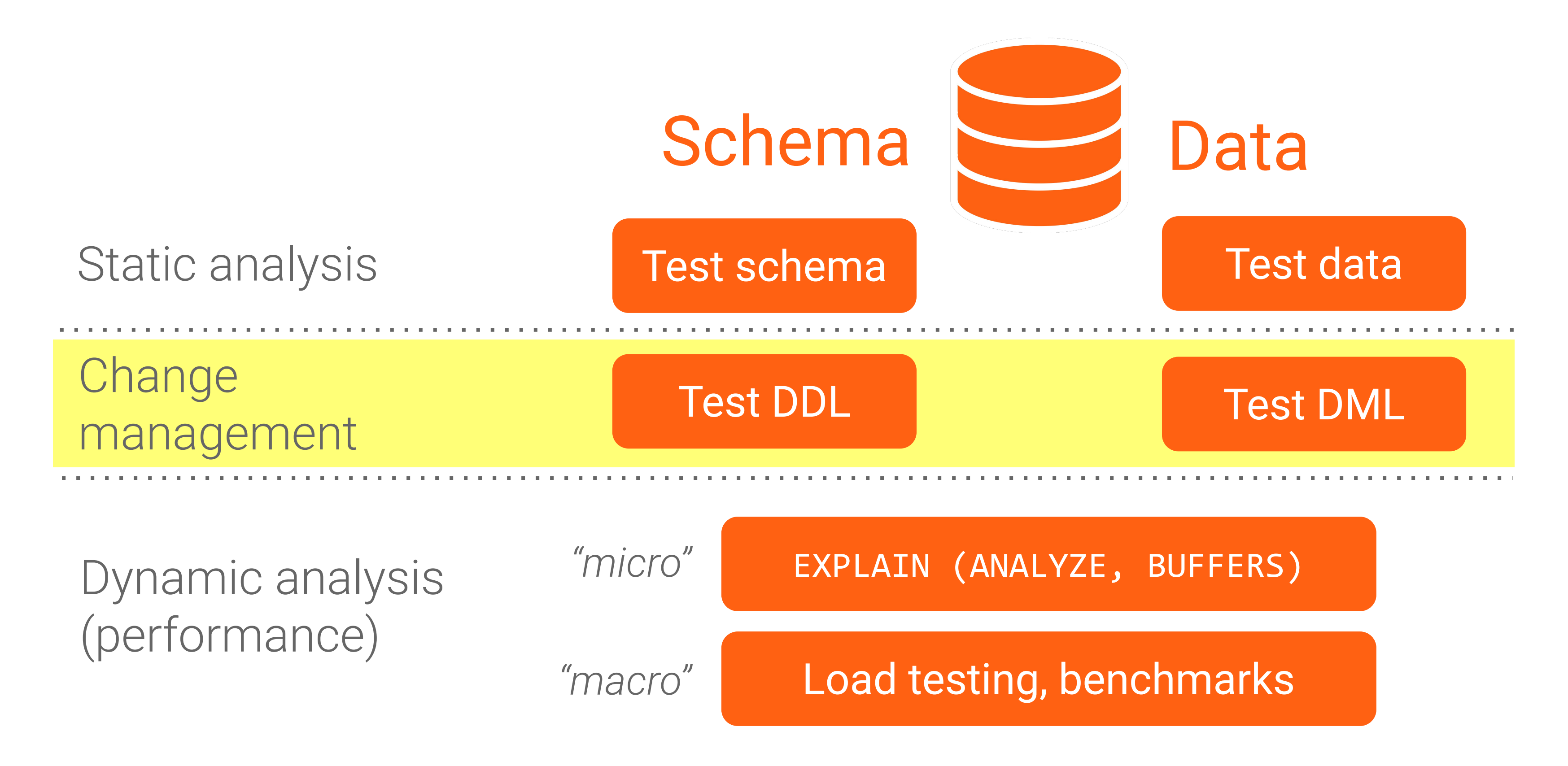
❌ Don’t:

- `IF [NOT] EXIST`

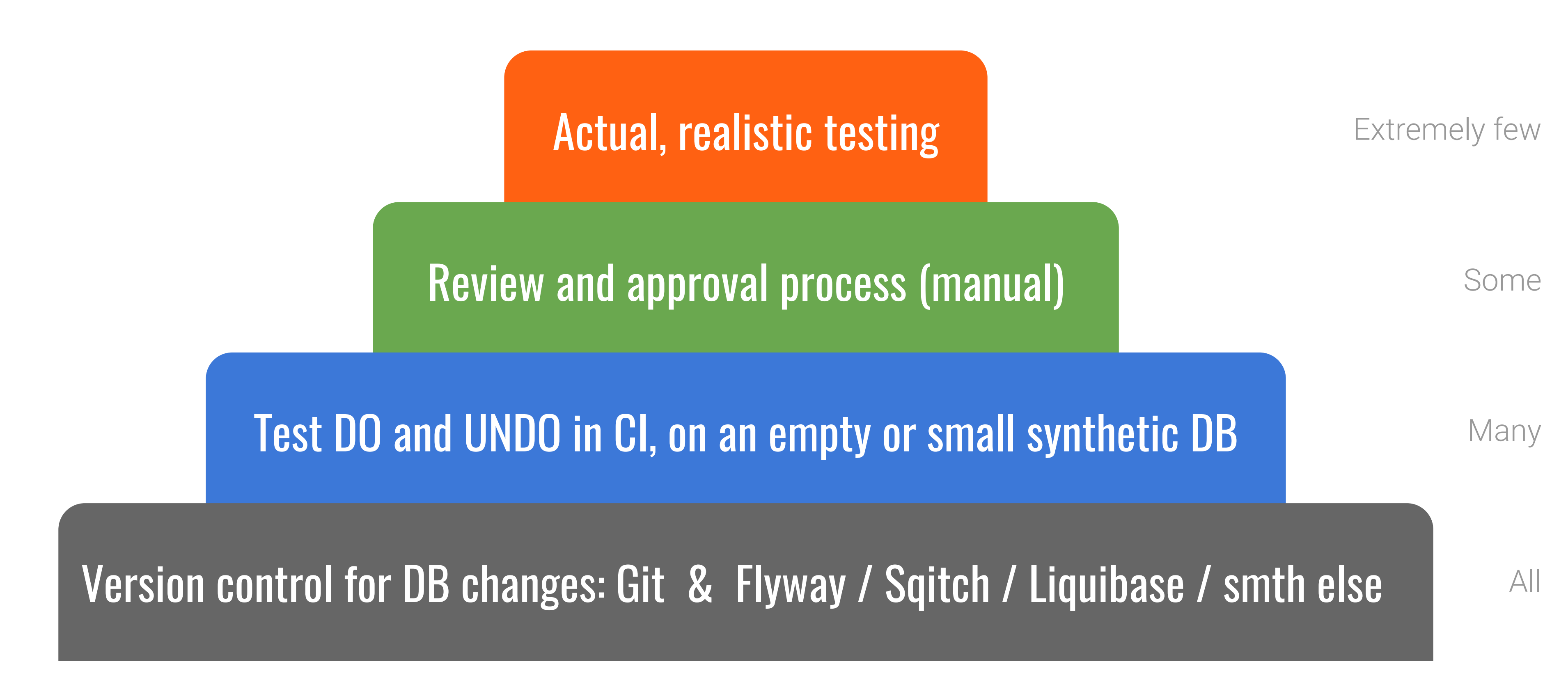

Do:

- test DO-UNDO-DO in CI
- keep schema up to date in all envs
- don’t ignore or work-around errors

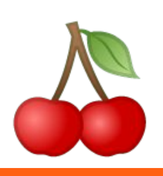
The Landscape of the Database Testing (app dev)



Reliable database changes – the hierarchy of needs



Postgres.ai



Actual, realistic testing — Extremely few

Review and approval process (manual) — Some

Test DO and UNDO in CI, on an empty or small synthetic DB — Many

Version control for DB changes: Git & Flyway / Sqitch / Liquibase / smth else — All

Example #2

You 2021-05-16 11:29:58

```
exec create table t1 as
  select id::int, random()::text as val
  from generate_series(1, 10000000) id;

alter table t1 add primary key (id);
```



Joe Bot 2021-05-16 11:29:59

```
exec create table t1 as select id::int, random()::text as val from generate_series(1, 10000000) id; alter table t1 add primary key (id);
```

Session: `webui-i4038`

```
% time      seconds wait_event
------ ------------ -----------------------------
64.82      9.447511 Running
7.92       1.154220 LWLock.WALWriteLock
6.94       1.011216 IO.DataFileExtend
5.69       0.829122 IO.WALWrite
5.27       0.767460 IO.WALSync
2.55       0.370954 IO.DataFileWrite
2.06       0.300581 IO.BufFileWrite
2.04       0.297535 IO.DataFileRead
1.51       0.220348 IO.DataFileImmediateSync
1.21       0.176163 IO.BufFileRead
------ ------------ -----------------------------
100.00    14.575110
```

The query has been executed. Duration: 14.575 s (estimated *for prod: 13.518...116.725 s)*
Estimated timing for production (experimental). How it works



Command

Postgres.ai

Example #2 – limited duration (15s)

You 2021-05-16 11:43:16

```
exec set statement_timeout to '15s'; update t1 set val = replace(val, '0159', 'OiSg');
```

Joe Bot 2021-05-16 11:43:16

```
exec set statement_timeout to '15s'; update t1 set val = replace(val, '0159', 'OiSg');
```

Session: `webui-i4038`

ERROR: ERROR: canceling statement due to statement timeout (SQLSTATE 57014)

❌ Failed

Example #2 – limited duration (15s)

You 2021-05-16 11:43:16

```
exec set statement_timeout to '15s'; update t1 set val = replace(val, '0159', 'OiSg');
```

Joe Bot 2021-05-16 11:43:16

```
exec set statement_timeout to '15s'; update t1 set val = replace(val, '0159', 'OiSg');
```

Session: `webui-i4038`

ERROR: ERROR: canceling statement due to statement timeout (SQLSTATE 57014)

❌ Failed

Example #2 – unlimited duration

You 2021-05-16 12:00:11

```
exec set statement_timeout to 0; update t1 set val = replace(val, '0159', 'OiSg');
```

Joe Bot 2021-05-16 12:00:12

```
exec set statement_timeout to 0; update t1 set val = replace(val, '0159', 'OiSg');
```

Session: `webui-i4038`

```
% time       seconds wait_event
------ ------------ ------------------------------
70.34      31.070133 Running
14.99       6.621164 LWLock.WALWriteLock
4.46        1.972113 IO.WALInitWrite
3.65        1.611055 IO.DataFileExtend
3.54        1.564610 IO.WALInitSync
1.38        0.608596 IO.WALWrite
1.33        0.588894 IO.DataFileRead
0.20        0.089901 LWLock.WALBufMappingLock
0.10        0.044417 IO.WALSync
------ ------------ ------------------------------
100.00     44.170883
```

The query has been executed. Duration: 44.171 s (estimated *for prod: 42.615...43.106 s)*
Estimated timing for production (experimental). How it works



Postgres.ai

Example #2 – unlimited duration

You 2021-05-16 12:00:11

```
exec set statement_timeout to 0; update t1 set val = replace(val, '0159', 'OiSg');
```

Joe Bot 2021-05-16 12:00:12

```
exec set statement_timeout to 0; update t1 set val = replace(val, '0159', 'OiSg');
```

Session: `webui-i4038`

```
% time        seconds wait_event
------ ------------ ------------------------------
70.34     31.070133 Running
14.99      6.621164 LWLock.WALWriteLock
4.46       1.972113 IO.WALInitWrite
3.65       1.611055 IO.DataFileExtend
3.54       1.564610 IO.WALInitSync
1.38       0.608596 IO.WALWrite
1.33       0.588894 IO.DataFileRead
0.20       0.089901 LWLock.WALBufMappingLock
0.10       0.044417 IO.WALSync
------ ------------ ------------------------------
100.00    44.170883
```

The query has been executed. Duration: 44.171 s (estimated *for prod: 42.615...43.106 s)*
Estimated timing for production (experimental). How it works

Completed

Heavy work later

Failed or blocked

Blocking others

Heavy work now

Example #2 – diagnostics: rows, buffers

```
test=# explain (buffers, analyze) update t1
         set val = replace(val, '0159', 'OiSg');
```

```
                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------
 Update on t1  (cost=0.00..189165.00 rows=10000000 width=42) (actual time=76024.507..76024.508 rows=0 loops=1)
   Buffers: shared hit=60154265 read=91606 dirtied=183191 written=198198
   ->  Seq Scan on t1  (cost=0.00..189165.00 rows=10000000 width=42) (actual time=0.367..2227.103 rows=10000000 loops=1)
         Buffers: shared read=64165 written=37703
 Planning:
   Buffers: shared hit=17 read=1 dirtied=1
 Planning Time: 0.497 ms
 Execution Time: 76024.546 ms
(8 rows)

Time: 76030.399 ms (01:16.030)
```

hit: ~459 GiB
read: ~716 MiB
dirtied: ~1.4 GiB
written: ~1.5 GiB

(with awful PG default settings)

Postgres.ai

Example #2 – UPDATEs vs. Bloat

```
test=# create table a1 as select 1::int as i;
SELECT 1

test=# select ctid, * from a1;
 ctid  | i
-------+---
 (0,1) | 1
(1 row)

test=# update a1 set i = i;
UPDATE 1
test=# select ctid, * from a1;
 ctid  | i
-------+---
 (0,2) | 1
(1 row)
```

Example #2 – what to do

Reduce the scope of work:

- Split to batches
- Temporary index to speed up lookups
- Avoid useless, silly updates

Avoid locking longer than 1s

Control dead tuples / bloat

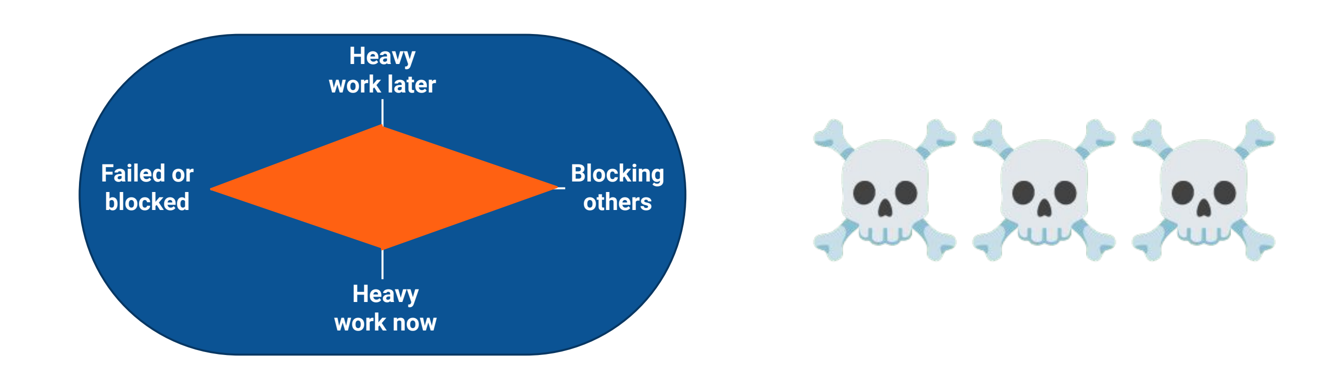


Example #3 – int4 PK problem

```
test=# insert into t1 select 2^31, '';
ERROR:  integer out of range
```

Example #3 – naïve method

```
test=# alter table t1 alter column id type int8;
ALTER TABLE
Time: 273726.829 ms (04:33.727)
```



Example #3 – ways to solve int4 PK problem

Avoid:

1a) Stop writing to the table

1b) Use negative values – another space of 2^31-1 values

Transform without downtime:

2a) “New column” method

2b) “New table” method

Example #3 – The “New column” method

- Create a int8 column
- Install a trigger to copy value for all fresh rows
- Backfill the values for the existing rows
- Redefine PK ———— a PK needs two things:
    - A unique index
    - NOT NULL constraint

      ☝️ *both these are not trivial*

- Finally, all FKs referring to the old PK need to be redefined

Example #3 – The “New column” method

How to create a unique index without downtime:

```
create unique index concurrently on tbl(new_int8_column);
```

Example #3 – The “New column” method

How to create a unique index without downtime:

```
create unique index concurrently on tbl(new_int8_column);
```

- might fail – it’s normal
- if failed, leaves an INVALID index behind
- cleanup & retry logic is needed
  (but not `DROP IF EXISTS`)

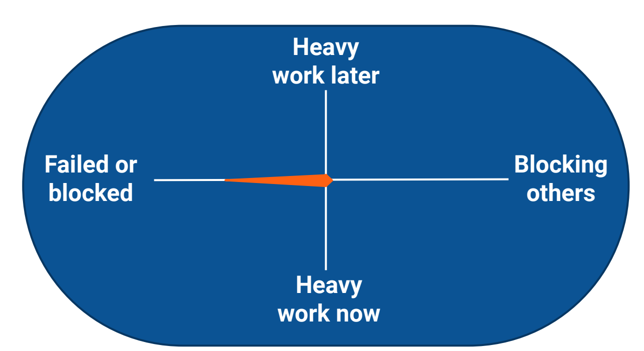
Postgres.ai

Example #3 – The "New column" method

How to create a unique index without downtime:

```
create unique index concurrently on tbl(new_int8_column);
```

- might fail – it's normal
- if failed, leaves an INVALID index behind
- cleanup & retry logic is needed
  (but not `DROP IF EXISTS`)



Postgres.ai

Example #3 – The "New column" method

How to add `NOT NULL` without downtime?

❌ Before Postgres 11 – impossible without downtime
- `NOT NULL` constraint is not an "online" operation
- `CHECK (.. IS NOT NULL)` is not "enough" for a PK

✅ Postgres 11+ trick:
- `alter table ... add column .. not null default -1;`
- Then "fix" all the -1 values
- Finally, drop the `DEFAULT`



Example #3 – The “New table” method

- CDC: a trigger + “delta” table to keep track of changes (or logical replication)
- REPEATABLE READ and snapshot export to get the initial data
- Take care of the constraints, indexes and *all* FKs
  - Redefining a FK is also not trivial:
    add `NOT VALID` (and `VALIDATE` *after* switching)
  - It’s even more tricky: FKs should be `DISABLED` till after switching
- Switch from the old table to the new one
  - in a single transaction
  - catching up the CDC “tail” inside the transaction



Final example – chain of blockers

Session 1:

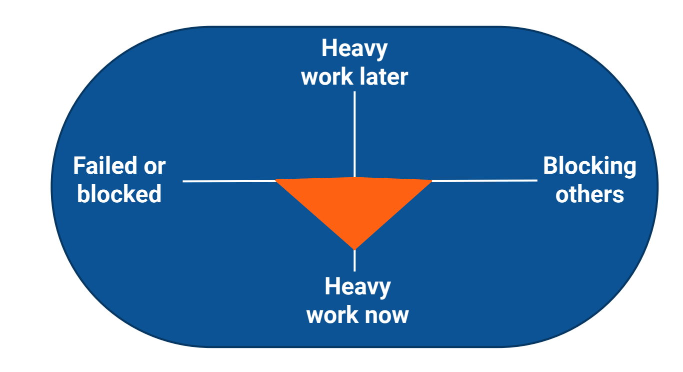
```
begin; update t1 set id = id where id = 1; -- and sit waiting
```

Session 2:

```
alter table t1 add column one_more int8;
```

Session 3:

```
select * from t1 where id = 2; -- boom!
```



Final example – chain of blockers

```
 change_age |  pid  | wait_event_type | wait_event | blocked_by_pids | state  | lvl | blocking_others |              latest_query_in_tx
------------+-------+-----------------+------------+-----------------+--------+-----+-----------------+-------------------------------------------
 00:06:41   | 28706 | Client          | ClientRead | {}              | idletx |   0 |               1 |  update t1 set id = id where id = 1;
 00:06:37   | 28709 | Lock            | relation   | {28706}         | active |   1 |               1 |  . alter table t1 add column one_more int8;
 00:06:28   | 28725 | Lock            | relation   | {28709}         | active |   2 |               0 |  .. select * from t1 where id = 2;
(3 rows)
```

"Forest of lock trees" https://gitlab.com/-/snippets/1890428



Ideal ALTER: lock_timeout & retries – use pl/pgsql

```
perform set_config('lock_timeout', lock_timeout, false); -- 50ms or so

for i in 1..max_attempts loop
  begin
    execute 'alter table t1 add column n1 int8';
    ddl_completed := true;
    exit;
  exception when lock_not_available then
    raise notice 'ALTER attempts: #% failed', i;
  end;
end loop;
```

Heavy work later

Failed or blocked

Blocking others

Heavy work now

How to run short ALTER TABLE
without long locking concurrent queries

https://www.depesz.com/2019/09/26/how-to-run-short-alter-table-without-long-locking-concurrent-queries/

(see the comment by *Mikhail Velikikh*)

How to become a “pro”

1. **Test *everything***

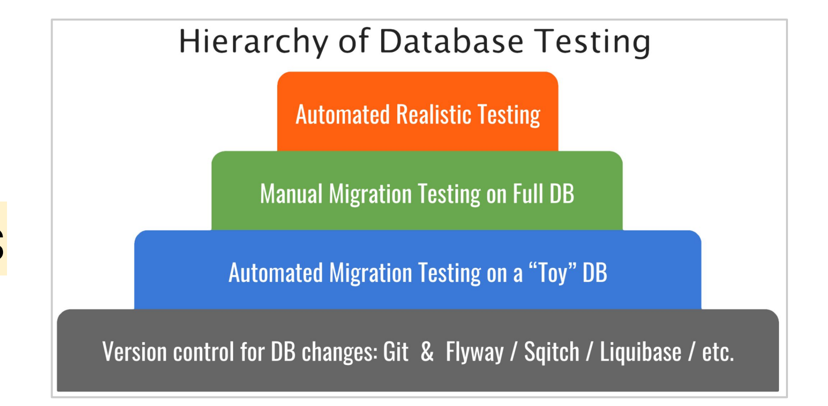
How to become a “pro”

1. **Test *everything***
2. **Make testing convenient**

Database Migration Testing with Database Lab

- Realistic migration testing is hard

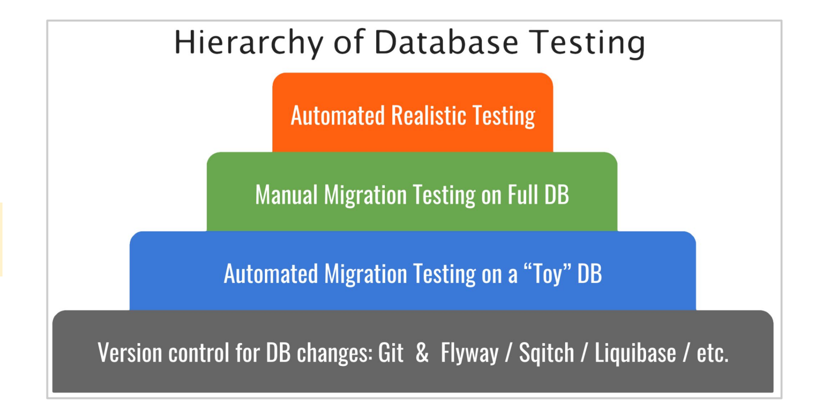
- No testing = unexpected problems



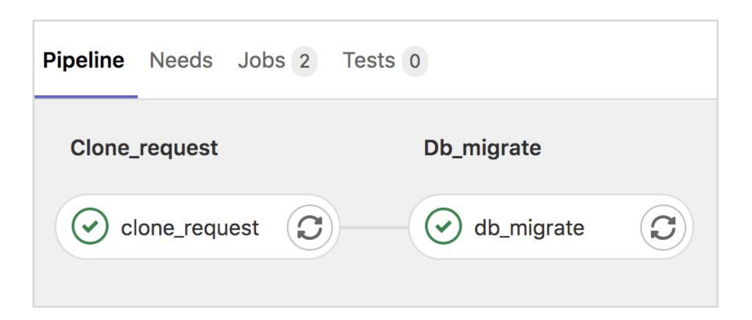
Database Migration Testing with Database Lab

- Realistic migration testing is hard

- No testing = unexpected problems



- Database Lab makes realistic testing *easy*

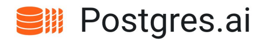


Thank you!

Slack (EN): slack.postgres.ai

Telegram (RU): t.me/databaselabru

Join the Database Lab *Customer Advisory Group:*
https://postgres.ai/customer-advisory-group



TO BE
CONTINUED....

Some examples of failures due to lack of testing

- Incompatible changes – production has different DB schema than dev & test
- Cannot deploy – hitting `statement_timeout` – too heavy operations

- During deployment, we've got a failover
- Deployment lasted 10 minutes, the app was very slow (or even down)

- Two weeks after deployment, we realize that the high bloat growth we have now has been introduced by that deployment
- Deployment succeeded, but then we have started to see errors



We need better tools

SCIENTIFIC AMERICAN



BICYCLE TECHNOLOGY

ONE DOLLAR

March 1973

COST OF TRANSPORT (CALORIES PER GRAM PER KILOMETER)

100
80
60
40
20
10
8
6
4
2
1
.8
.6
.4
.2
.1

MICE
LEMMING
BLOWFLY
FRUIT FLY
BEE
RAT
LOCUST
HUMMINGBIRD
RABBIT
HELICOPTER
BUDGERIGAR
JET FIGHTER
GULL
DOG
LIGHT PLANE
PIGEON
SHEEP
COW
AUTOMOBILES
MAN
JET TRANSPORT
HORSES
SALMON
MAN ON BICYCLE

10^{-6} 10^{-5} 10^{-4} 10^{-3} 10^{-2} 10^{-1} 1 10 10^{2} 10^{3} 10^{4} 10^{5} 10^{6}

BODY WEIGHT (KILOGRAMS)

Steve Jobs (1980)

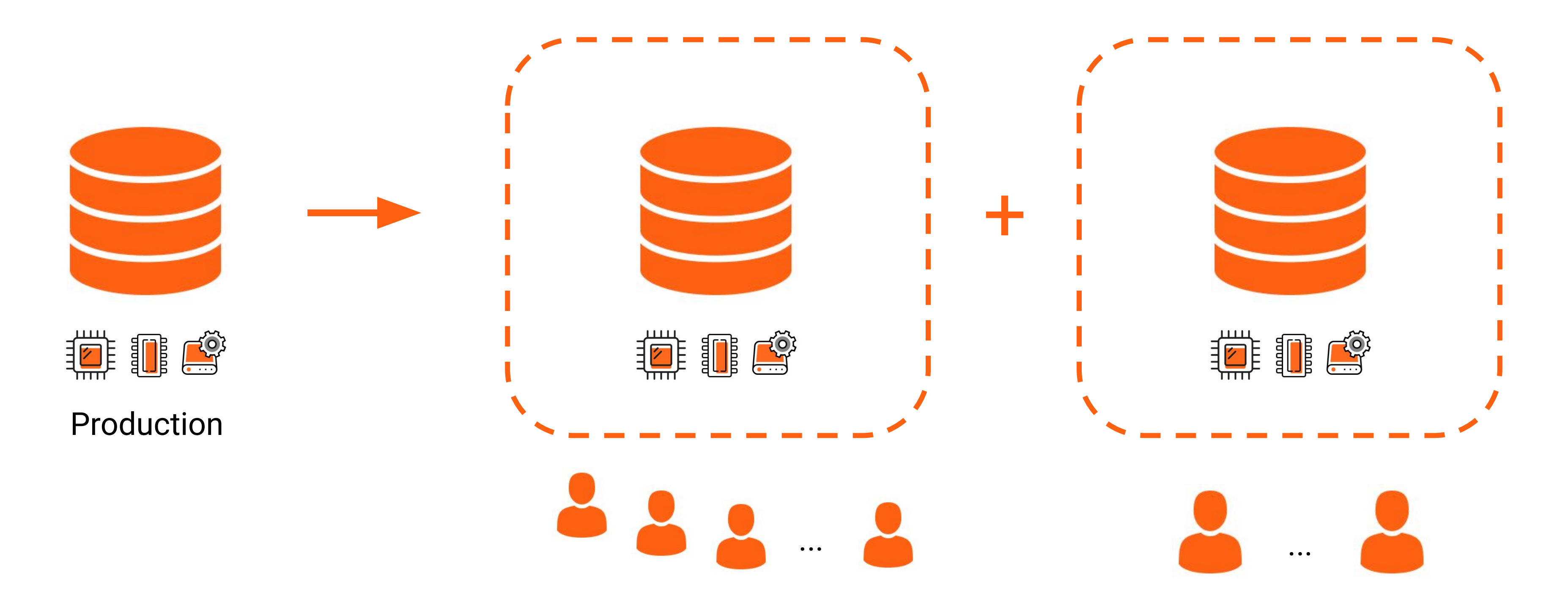


1) We, humans, are great tool-makers.
   We amplify human abilities.

2) Something special happens
   when you have 1 computer and 1 person.

   It’s very different that having 1 computer and 10 persons.

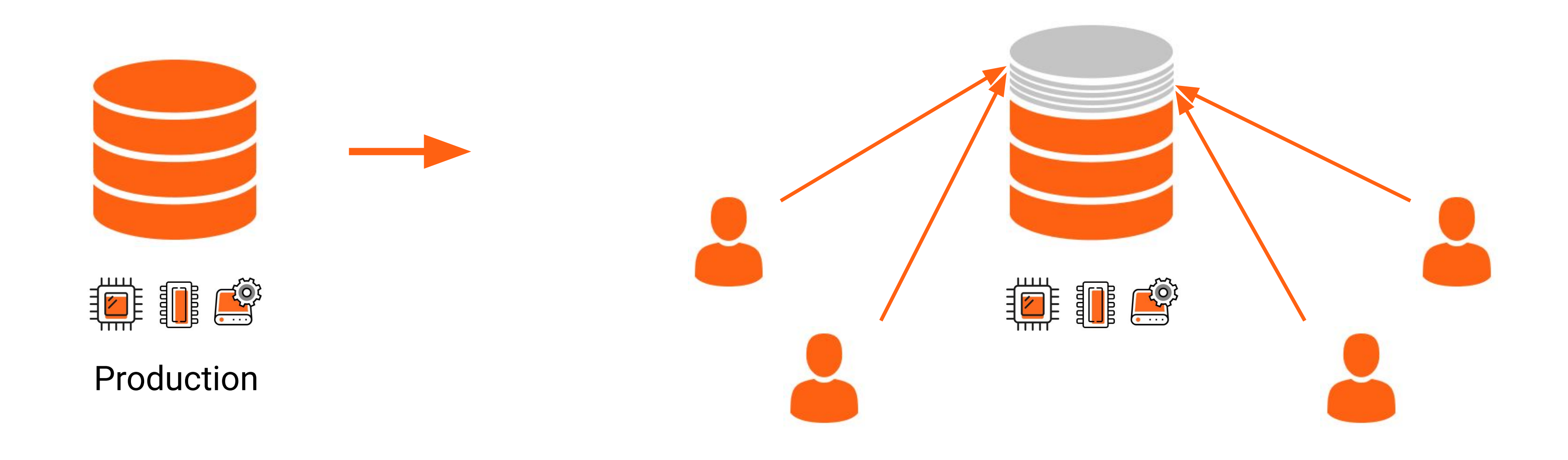
Traditional DB experiments – **thick** clones



“1 database copy – 10 persons”

Postgres.ai

Database Lab: use *thin* clones



"1 database copy – 1 person"

“Thin clones” – Copy-on-Write (CoW)

Shared data blocks

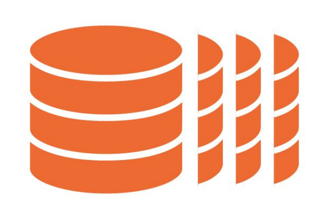
Extra blocks for changes

Thick copy of production (any size)

Thin clone (size starts from 1 MB, depends on changes)



Database Lab – Open-core model



The Database Lab Engine (DLE)

Open-source (AGPLv3)

- Thin cloning – API & CLI
- Automated provisioning and data refresh
- Data transformation, anonymization
- Supports managed Postgres (AWS RDS, etc.)

https://gitlab.com/postgres-ai/database-lab

The Platform (SaaS)

Proprietary (freemium)

- Web console – GUI
- Access control, audit
- History, visualization
- Support

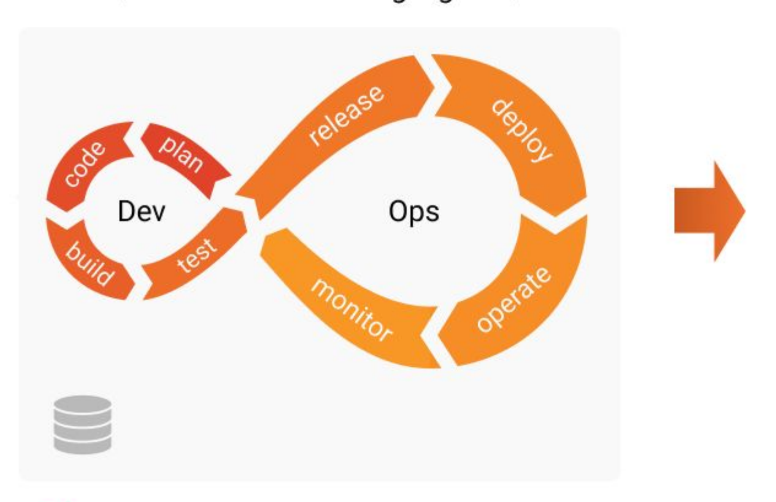
https://postgres.ai/

^^ use these links to start using it for your databases ^^

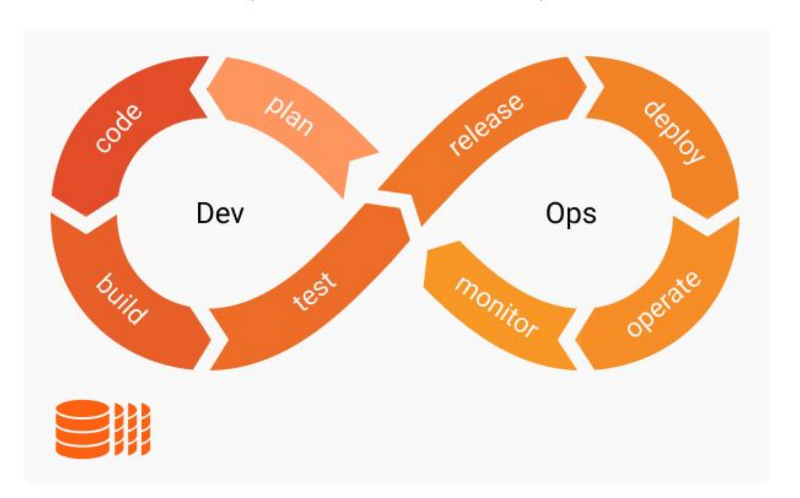


Database Lab unlocks “Shift-left testing”

Development bottlenecks
(with standard staging DB)



Frictionless development
(with Database Lab)



- ✗ Bugs: difficult to reproduce, easy to miss
- ✗ Not 100% of changes are well-verified
- ✗ SQL optimization is hard
- ✗ Each non-prod big DB costs a lot
- ✗ Non-prod DB refresh takes hours, days, weeks

- ✓ Bugs: easy to reproduce, and fix early
- ✓ 100% of changes are well-verified
- ✓ SQL optimization can be done by anyone
- ✓ Non-prod DB refresh takes seconds
- ✓ Extra non-prod DBs doesn't cost a penny

Database experiments on thin clones – yes and no

Yes

- Check execution plan – Joe bot
  - EXPLAIN w/o execution
  - EXPLAIN (ANALYZE, BUFFERS)
    - (timing is different; structure and buffer numbers – the same)
- Check DDL
  - index ideas (Joe bot)
  - auto-check DB migrations (CI Observer)
- Heavy, long queries: analytics, dump/restore
  - No penalties!
    (think hot_standby_feedback, locks, CPU)

No

- Load testing
- Regular HA/DR goals
  - backups
    - (but useful to check WAL stream, recover records by mistake)
  - hot standby
    - (but useful to offload very long-running SELECTs)



DB migration testing – “stateful tests in CI”

What we want from testing of DB changes:

- Ensure the change is valid
- It will be executed in appropriate time
- It won’t put the system down

…and:

- What to expect? (New objects, size change, duration, etc.)

Perfect Lab for database experiments

- Realistic conditions – as similar to production as possible
  - The same schema, data, environment as on production
  - Very similar background workload
- Full automation
- “Memory” (store, share details)
- Low iteration overhead (time & money)
- Everyone can test independently

  allowed to fail → allowed to learn



Database experiments with Database Lab today (2021)

- Realistic conditions – as similar to production as possible
  - The same schema, data, environment as on production
  - ~~Very similar background workload~~
- Fine automation
- "Memory" (store, share details)
- Low iteration overhead (time & money)
- Everyone can test independently

  able to fail → able to learn



Why Database Lab was created

- Containers, OverlayFS (file-level CoW)

  CI: `docker pull ... && docker run ...`

  – OK only for tiny (< a few GiB) databases

- Existing solutions: Oracle Snap Clones, Delphix, Actifio, etc.
  $$$$, not open

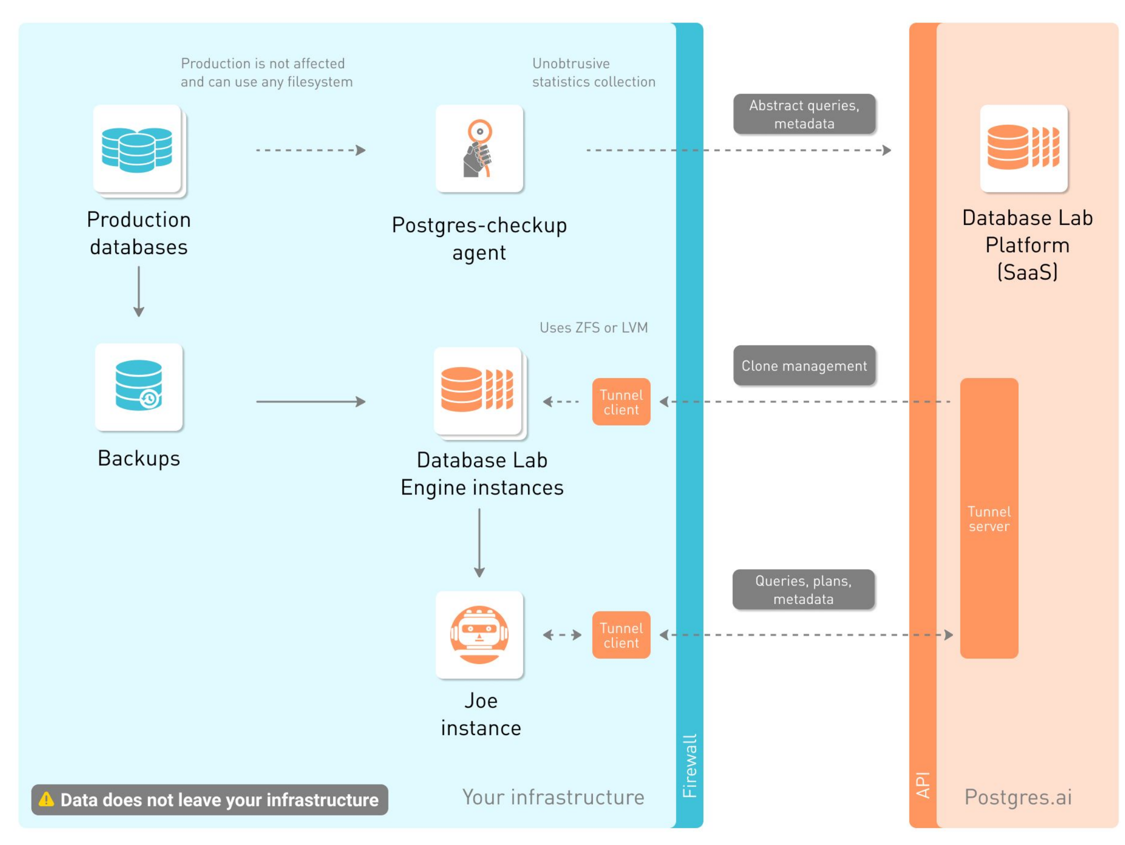
  – OK only for very large enterprises



Companies that do need it today

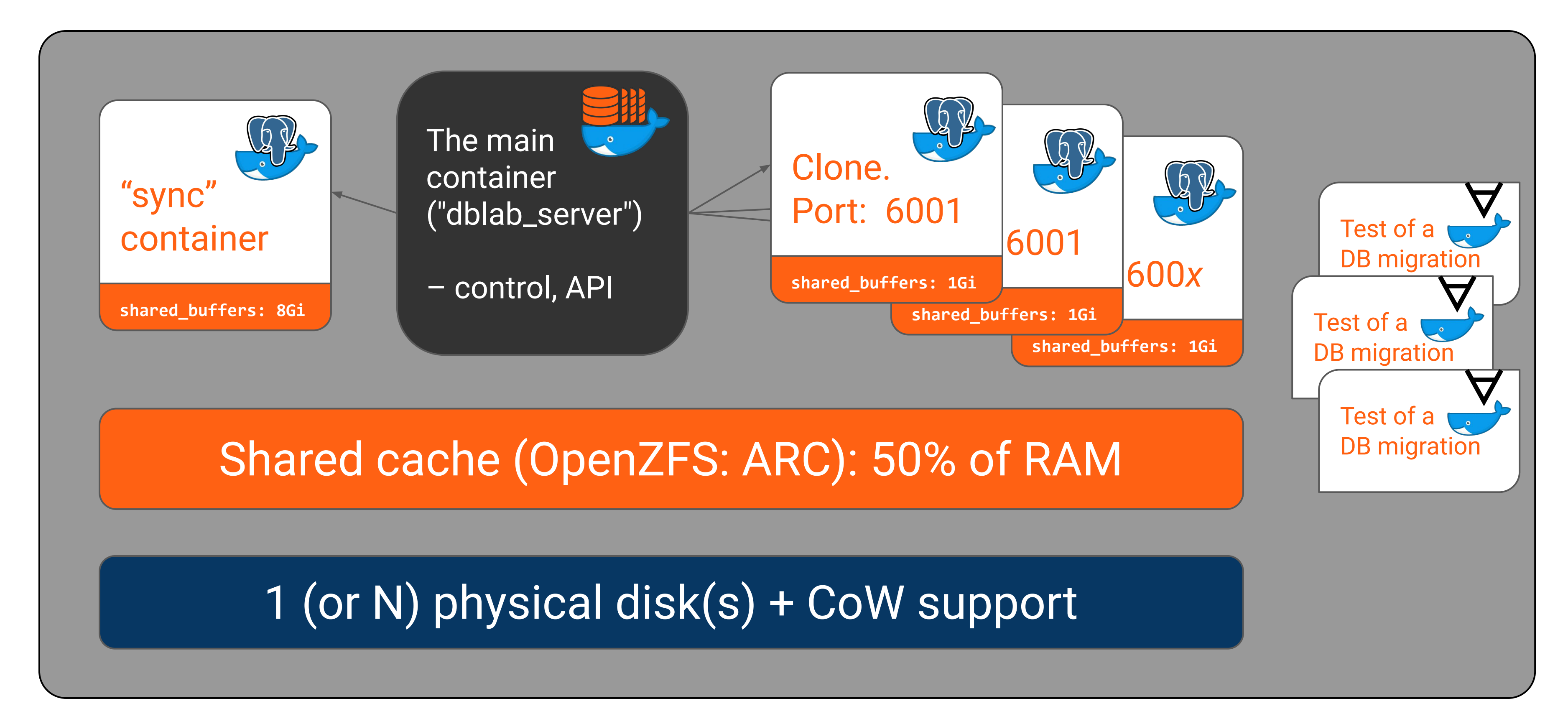
- 10+ engineers
- Multiple backend teams (or plans to split soon)
- Microservices (or plans to move to them)
- 100+ GiB databases
- Frequent releases

Database Lab – a high-level overview (with SaaS)

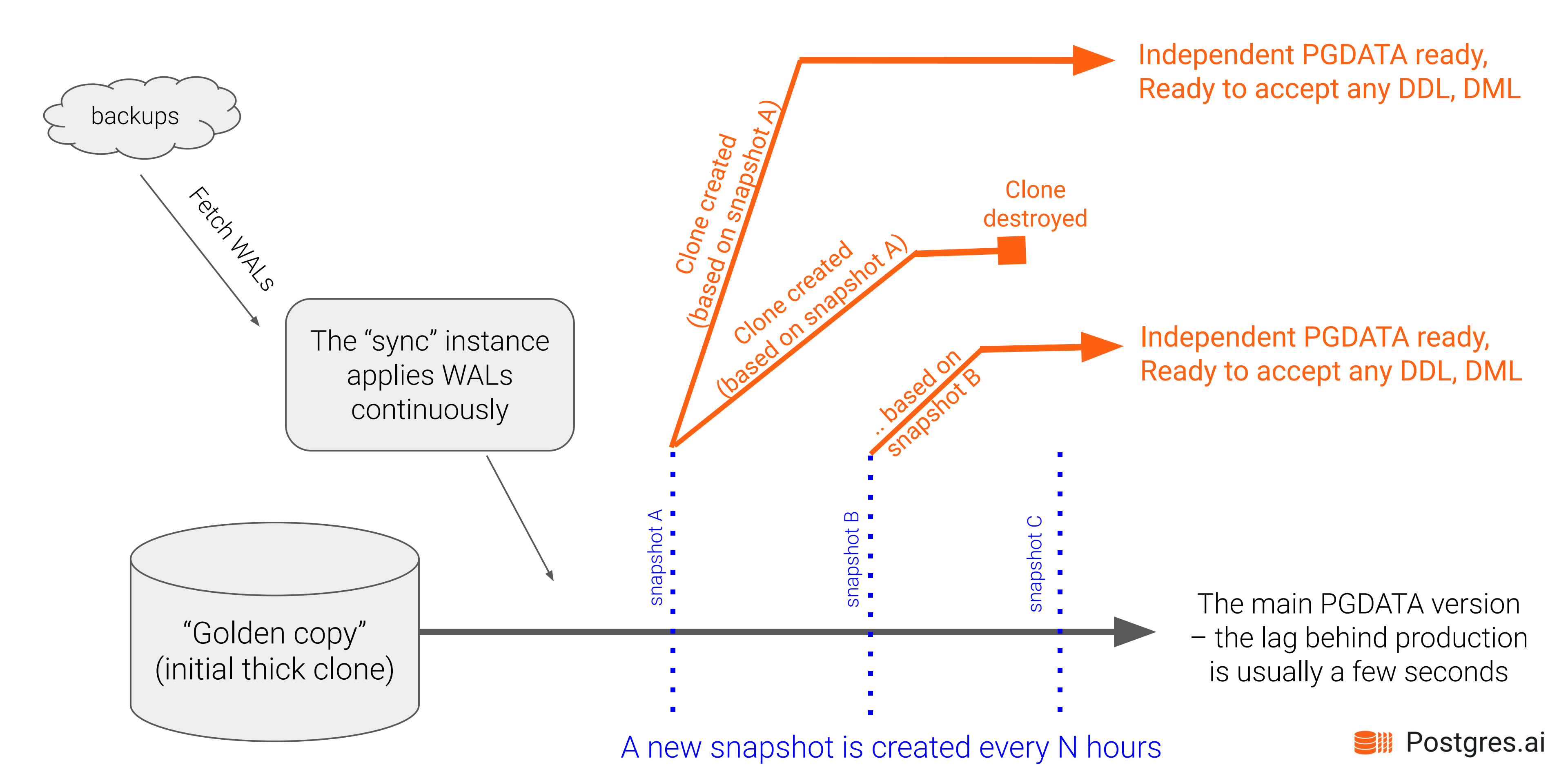


Postgres.ai

Inside the Database Lab Engine 2.x



DLE – the data flow (physical mode)



How snapshots are created (ZFS version)

- Create a “pre” ZFS snapshot (R/O)
- Create a “pre” ZFS clone (R/W)
- DLE launches a temporary “promote” container
  - If needed, performs “preprocessing” steps (bash)
  - Uses “pre” clone to run Postgres and promote it to primary state
  - If needed, performs “preprocessing” SQL queries
  - Performs a clean shutdown of Postgres
- Create a final ZFS snapshot that will be used for cloning



Major topics of automated (CI) testing on thin clones

- Security

  https://postgres.ai/docs/platform/security

- Capturing dangerous locks

  CI Observer: https://postgres.ai/docs/database-lab/cli-reference#subcommand-start-observation

- Forecast production timing

  Timing estimator: https://postgres.ai/docs/database-lab/timing-estimator

Making the process secure: where to place the DLE?

PII here

Production

The big wall

Dev & Test

CI runners here

Where to place the DLE? Current approach

PII here

Production

The big wall

Dev & Test

CI runners here

High-level API calls

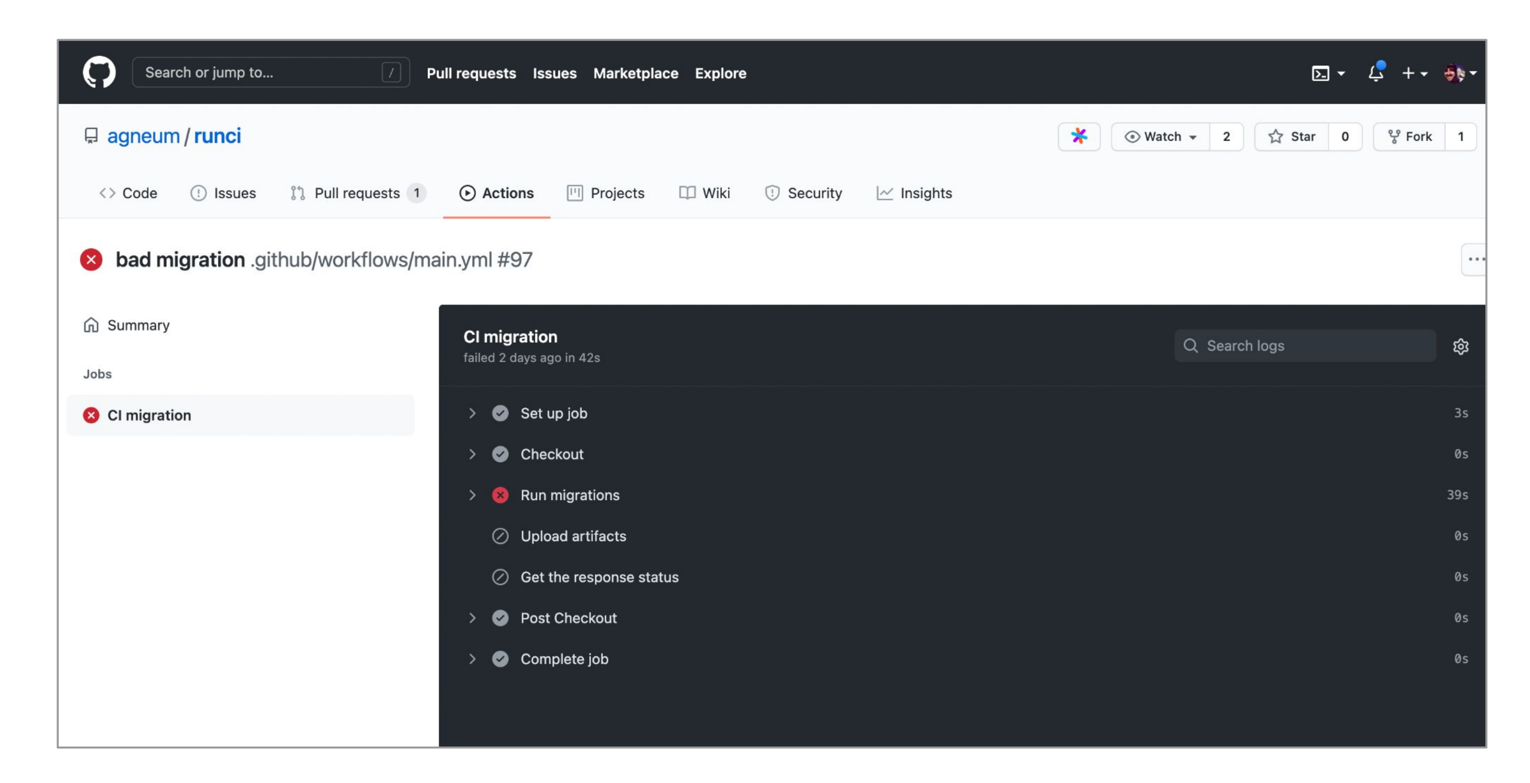






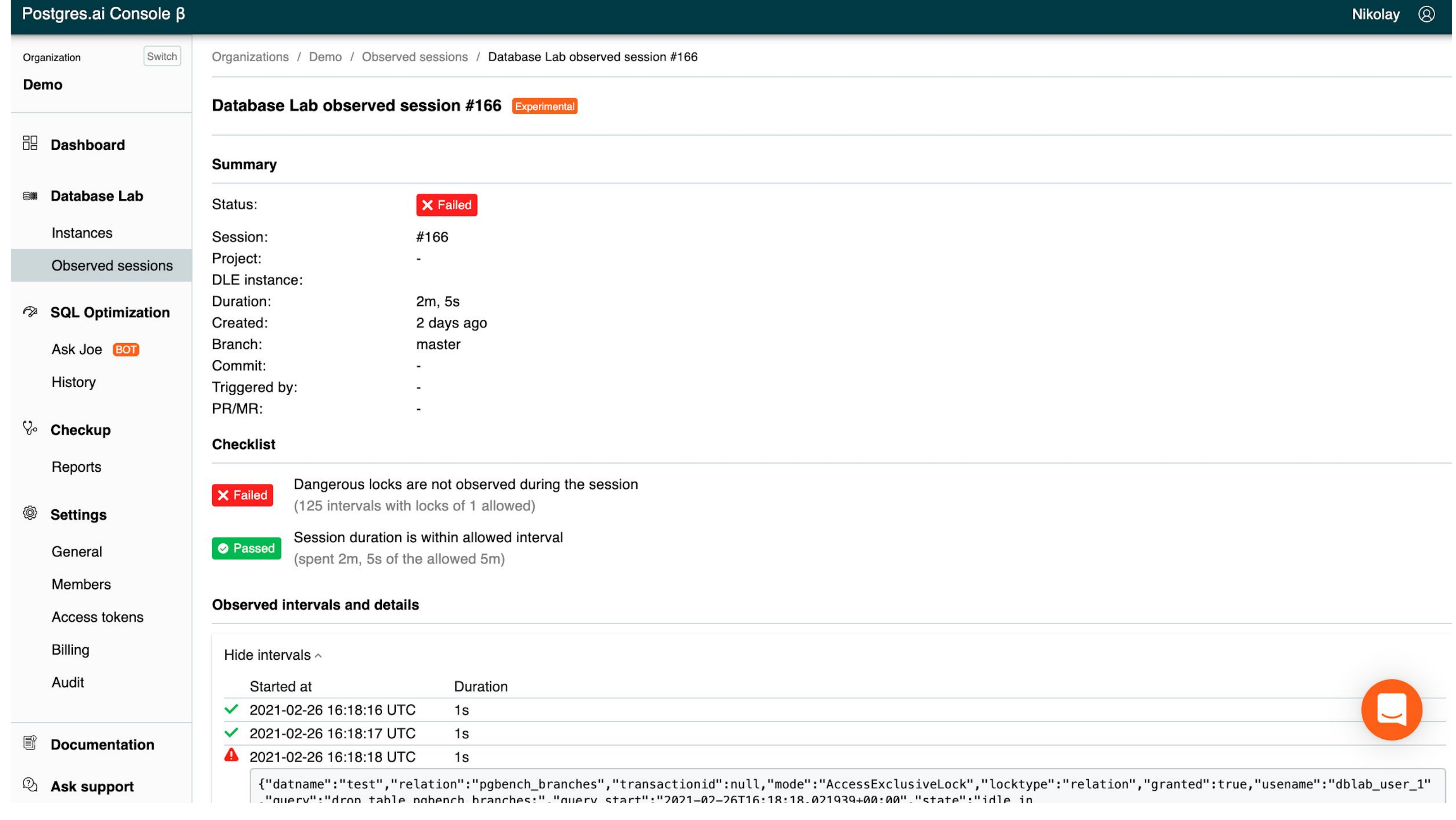
How it looks like: CI part

Example: GitHub Actions:
https://github.com/agneum/runci/runs/2519607920?check_suite_focus=true

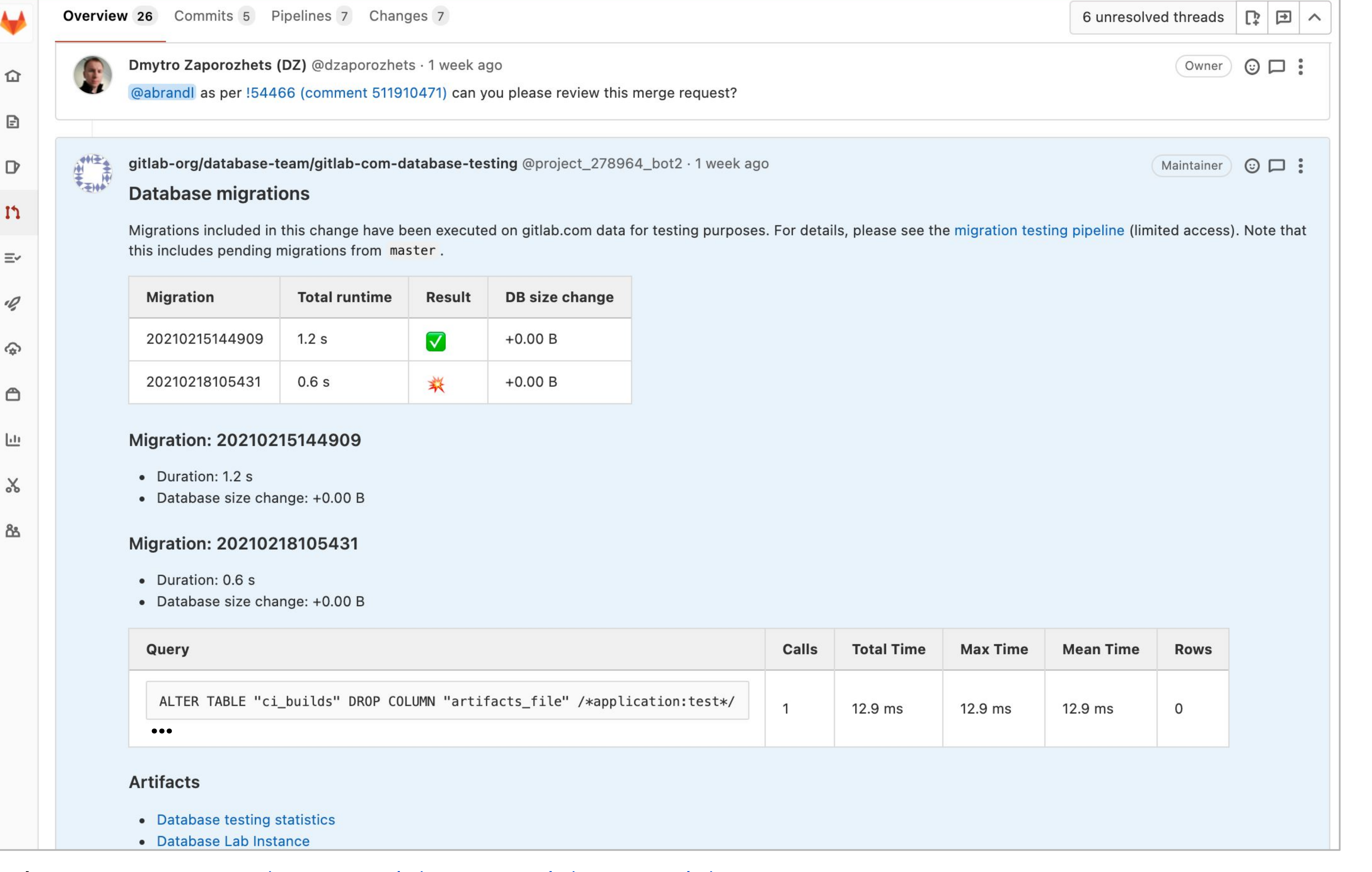


Postgres.ai

More about dangerous lock detection



Postgres.ai



February 19, 2021 – https://gitlab.com/gitlab-org/gitlab/-/merge_requests/54564#note_512678910

Postgres.ai

Example: GitLab.com, testing database changes using Database Lab

- Full automation
- GitLab CI/CD pipelines securely work with Database Lab
- Database Lab clones ~10 TiB database in ~10 seconds

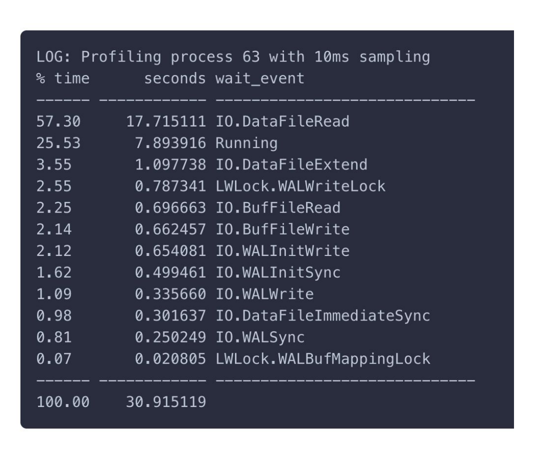
Read their blueprint:

https://docs.gitlab.com/ee/architecture/blueprints/database_testing/

More about production timing estimation

Experimental, WIP: https://postgres.ai/docs/database-lab/timing-estimator

```
estimator:
   readRatio: 1
   writeRatio: 1
   profilingInterval: 20ms
   sampleThreshold: 100
```



```
LOG: Profiling process 63 with 10ms sampling
% time      seconds wait_event
------ ------------ -----------------------------
57.30     17.715111 IO.DataFileRead
25.53      7.893916 Running
3.55       1.097738 IO.DataFileExtend
2.55       0.787341 LWLock.WALWriteLock
2.25       0.696663 IO.BufFileRead
2.14       0.662457 IO.BufFileWrite
2.12       0.654081 IO.WALInitWrite
1.62       0.499461 IO.WALInitSync
1.09       0.335660 IO.WALWrite
0.98       0.301637 IO.DataFileImmediateSync
0.81       0.250249 IO.WALSync
0.07       0.020805 LWLock.WALBufMappingLock
------ ------------ -----------------------------
100.00    30.915119
```



Summary:

```
Time: 3.148 s
  - planning: 0.168 ms
  - execution: 3.147 s (estimated* for prod: 2.465...2.693 s)
    - I/O read: 627.267 ms
    - I/O write: 3.644 ms

Shared buffers:
  - hits: 1016393 (~7.80 GiB) from the buffer pool
  - reads: 16395 (~128.10 MiB) from the OS file cache, including disk I/O
  - dirtied: 16395 (~128.10 MiB)
  - writes: 280 (~2.20 MiB)
```

Summary – available in PR/MR and visible to whole team

- When, who, status
- Duration (in the Lab + estimated for production)
- Size changes, new objects
- Dangerous locks
- Error stats
- Transaction stats
- Query analysis summary
- Tuple stats
- WAL generated, checkpoitner/bgwriter stats
- Temp files stats

Example (WIP): https://gitlab.com/postgres-ai/database-lab/-/snippets/2083427

More artifacts, details – restricted access

- System monitoring (resources utilization)
- pg_stat_*
- pg_stat_statements, pg_stat_kcache
- logerrors
- Postgres log
- pgBadger (html, json)
- wait event sampling
- perf tracing, flamegraphs; or eBPF
- Estimated production timing

https://gitlab.com/postgres-ai/database-lab/-/issues/226



Database Lab Roadmap

https://postgres.ai/docs/roadmap

- Lower the entry bar
  - Simplify installation
  - Simplify the use
  - Easy to integrate
  - *** **** * *******

Where to start

Postgres.ai/docs/

